A Fast Parallel Selection Algorithm on GPUs

Darius Bakunas-Milanowski*, Vernon Rego†, Janche Sang*, and Chansu Yu*
*Dept. of Electrical Engineering and Computer Science
Cleveland State University, Cleveland, OH, USA
†Dept. of Computer Science,
Purdue University, West Lafayette, IN, USA

***Abstract*—Today, parallel selection algorithms that run on Graphical Processing Units (GPUs) hold great promise in providing even more computational power than that of conventional CPUs. To quantify these gains, we examined a new parallel selection algorithm to see exactly what its vast number of simple, data parallel, multithreaded cores meant for performance times, using the current generation of NVIDIA GPUs. Specifically, our team tested how we could utilize a GPU to select elements from a massive array that met specific criteria and store their indices in a target array for additional processing. In this paper, we report optimization techniques and road blocks encountered. Overall, the experimental results demonstrate that our implementation performs an average of 3.67 times faster than Thrust, an open-source parallel algorithms library.**

***Keywords*-Parallel Selection; CUDA Thrust Library; GPU; Optimization Techniques; SIMT**

I. INTRODUCTION

Selection, also known as stream compaction or filtering, is a common programming concept that has a wide range of applications in the area of statistics, database software, artificial intelligence, image processing, and simulations [1][2][3]. It produces a smaller output array, containing only wanted elements from the input array made up of the mixed elements. With the tremendous amount of data elements to be processed, better performance becomes a key factor in implementing these algorithms. Therefore, exploiting the availability and the power of multiprocessors to speed up the execution is of considerable interest.

In the past few years, modern Graphics Processing Units (GPUs) have been increasingly used together with CPUs to accelerate a broad array of scientific computations in so-called heterogeneous computing[4]. It is now much more convenient to create application software that will run on current GPUs for processing massively large amounts of data, without the need to write low-level assembly language code. Furthermore, a selection of accelerated, high performance libraries allows an easy way of adding GPU-acceleration to the wide array of scientific applications. One can get even more flexibility and speed by writing his or her own GPU-accelerated programs using the CUDA Thrust Library[5], which provides a comprehensive development environment for C and C++ developers.

As shown in Figure 1, NVIDIA Kepler GPUs consist of a scalable number of streaming multiprocessors (SMXs), each containing a group of streaming processors (SPs) or cores to execute the light-weighted threads, warp by warp, using the Single Instruction, Multiple Threads (SIMT) style (term coined by NVIDIA manufacturer).

In addition to the main memory on the CPU motherboard, the GPU device has its own off-chip device memory (i.e. global memory). The kernel function, which is executed on the device, is composed of a grid of threads. Note that a grid is divided into a set of blocks and each block contains multiple warps of threads. Blocks are distributed evenly to the different SMXs to run. Furthermore, registers and shared memory in a SMX are on-chip memory and can be accessed very fast. They are per-block resources and are not released until all the threads in the block finish execution. Each SMX also has 32 special function units (SFUs) for fast approximate transcendental operations, like __cosf(), __expf(), etc. and 32 load/store (LD/ST) units for memory read/write operations.

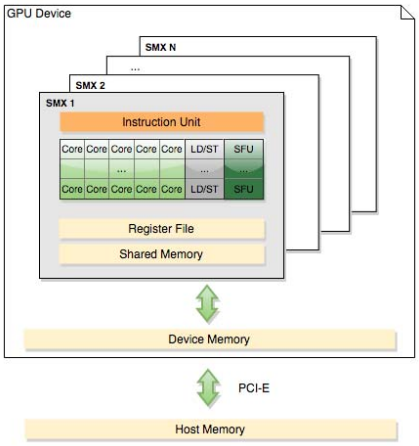

Figure 1: Kepler GPU block diagram

In this paper, we focused on the design and implementation of a new parallel selection algorithm by using Kepler's shuffle instructions which allow threads within a warp to exchange data. We also compared its performance with other parallel selection methods on CUDA-enabled

978-1-4673-9795-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CSCI.2015.132

GPUs. All tests were performed using CUDA Toolkit on a PC with a consumer grade NVIDIA GeForce GTX 770 GPU and also on the Ohio Supercomputer Center's newest cluster Ruby, outfitted with professional NVIDIA Tesla K40 GPUs. Both of these cards belong to the NVIDIA Kepler[6], a cutting-edge high performance computing architecture. The empirical results show that our algorithm, which also preserve the relative order of the input elements, performs much faster than the Thrust library.

The organization of this paper is as follows. Section 2 describes related work. Section 3 goes in to details of our implementation and finally, in Section 4, the experiments and the results for performance evaluation are presented. We give a short conclusion in Section 5.

II. RELATED WORK

Sequential selection is a common function and it is available in many programming languages and/or libraries. However, to implement the parallel selection, the challenging is how to determine the indices of the selected elements in the destination array. In general, the approaches to performing the stream compaction on multiprocessors can be classified into two categories: one is based on the atomic operation, while the other is based on the list ranking using the prefix-sum algorithm [7], as described below in detail.

A. Atomic operation based approaches

In the former approach, we can use an index counter, which will be incremented by one for each newly selected element. Since many threads share the counter, the addition has to be an atomic operation. This can be done by using CUDA atomicInc() function. Note that a CUDA atomic function performs a read-modify- write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

The main problem with this approach is that these atomic operations become a major bottleneck when the input contains a large amount of elements that pass our selection criteria. This is due to the very large number of threads competing to increment the single counter inside the global memory.

One possible improvement is to use shared memory atomics. This will essentially decrease the number of atomic collisions to a block size. Unfortunately, its performance still suffers from the thread synchronization. As demonstrated in the Experimental Results Section, execution time for both algorithms is directly proportional to the number of passing items.

A modified approach is discussed in the article "CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics"[8], written by researcher A. Adinetz. In addition to using the aggregated atomicAdd(), it uses the primitives __ballot(), __ffs() and __popc() (Compute Capability 2.0 and above) to perform intra-warp scan[9] and also uses the warp shuffle intrinsic[10] which is available on the Kepler and later GPUs (Compute Capability 3.0 and above) to broadcast the group index value to all of the threads within a warp. His implementation of atomic function with warp aggregation is isolated from the rest of the application, and can be used as a drop-in replacement for existing code that use CUDA atomics. Furthermore, since each warp will issue at most one atomicAdd() request, its execution time will not be proportional to the number of passing elements and hence can be reduced greatly. Note that these three algorithms based on atomic operation do not preserve the relative order of the input elements, thus might not be suitable for certain applications.

B. List ranking based approaches

For the latter approach, one such implementation is provided by the Thrust library, specifically a method called copy_if()[11], which is a fairly good implementation, simple to use and may prove to be the best choice for most users needing this operation. By digging into its implementation details, we found that it uses 2-level sums. First, it calculates the number of selected elements within the block (using index counter inside the shared memory) and stores the result in an intermediate array of size N / block_size (N is the total number of input elements) in the global memory. Then it performs a parallel prefix sum on this array and uses the outcome in the final phase to determine the indices of output elements. As a result, the relative order of the input elements is also preserved. Furthermore, as shown in the later section, Thrust algorithm execution time does not depend on the number of passing elements. Our previous work in [12] is also based on the list-ranking approach. It used the __ballot() and __popc() to find the offset of a selected element in a warp quickly. We also optimized the code to let each thread handle many elements to increase the ratio of computation/communication. It performs much faster than the Thrust library, but slower than the Adinetz algorithm.

III. A NEW ALGORITHM

The following algorithm implementation consists of three major phases. The phase 1 kernel starts by evaluating a predicate for each subgroup of 32 elements and saving the result into a predicate array (see Figure 2). The number of passing elements in the subgroup is also determined (using __popc() instruction) and saved inside the register variable cnt for each thread in the warp. This operation is performed

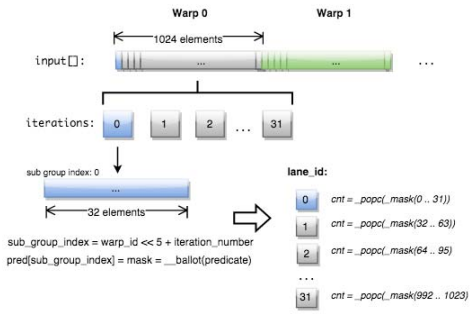

Figure 2: Phase 1a: fill predicate and counter arrays

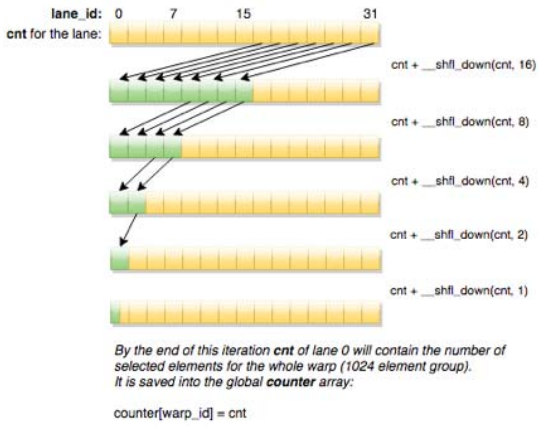

Figure 3: Phase 1b: perform reduction on cnt values

```
1 #include <cuda.h>
2 #include "algorithms.h"
3 #include "err.h"
4
5 #define WARP_SZ 32
6
7 __device__ inline int lane_id(void) { return threadIdx.x % WARP_SZ; }
8
9 __global__ void alg7_kernel_phase1(float *input,
10   unsigned int *counter,
11   unsigned int *pred,
12   float percent, const
13   unsigned int num_items)
14 {
15    int tid = blockIdx.x * blockDim.x + threadIdx.x;
16
17    if (tid >= (num_items >> 5) )  // divide by 32
18     return;
19
20    int lnid = lane_id();
21    int warp_id = tid >> 5; // global warp number
22
23    unsigned int mask;
24    int cnt;
25
26    for(int i = 0; i < 32 ; i++) {
27        mask = __ballot(input[(warp_id<<10)+(i<<5)+lnid] <= percent);
28
29        if (lnid == 0)
30            pred[(warp_id<<5)+i] = mask;
31
32        if (lnid == i)
33            cnt = __popc(mask);
34    }
35    // para reduction to a sum of 1024 elements
36 #pragma unroll
37    for (int offset = 16 ; offset > 0; offset >>= 1)
38        cnt += __shfl_down(cnt, offset);
39
40    if (lnid == 0)
41        counter[warp_id] = cnt; // store the sum of the group
42 }
43
```

Figure 4: Phase 1 kernel

for 32 iterations. As a result, each warp processes the total number of 1024 elements.

When this loop completes, a parallel reduction operation is performed on the cnt register values, resulting in the number of selected elements for each 1024-element group (see Figure 3). Note that __shfl_down() instruction was used, that essentially allows passing down register values from the lane with higher ID relative to the caller lane. The result is saved in the counter array. The detail of the code for the Phase 1 Kernel can be found in Figure 4.

In phase 2, a prefix sum operation is applied to the counter array (Figure 5). As a result, there are counter[k-1] valid elements before the group k. Note that for this operation we simply used Thrust implementation thrust::inclusive_scan(x), which was fast enough and sufficient for our purposes.

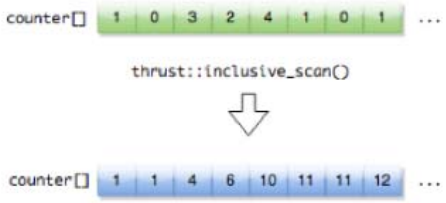

Figure 5: Phase 2: perform inclusive scan operation on the counter array

Phase 3 produces the final result. The process begins by reading the predicate array that was produced in the phase 1. The number of set bits is determined (using __popc() instruction) for each predicate value and saved into the cnt register for each lane inside the warp (Figure 6).

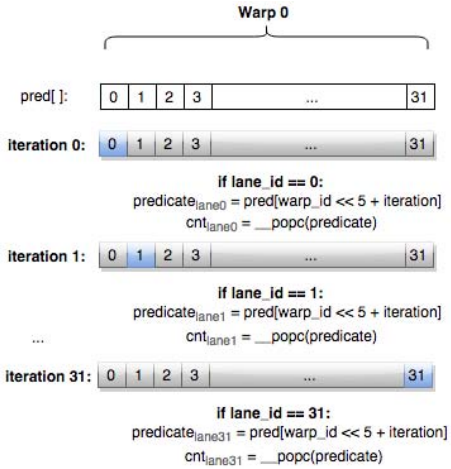

Figure 6: Phase 3a: get bit counts for the predicate values

To calculate the subgroup index, the prefix sum operation is performed on these cnt values (Figure 7). After this

operation, each cnt i (for thread i) holds the number of valid elements before the ith subgroup. Note, for this operation we used __shfl_up() instruction, which allows passing register values form the lane with lower ID relative to the caller lane.

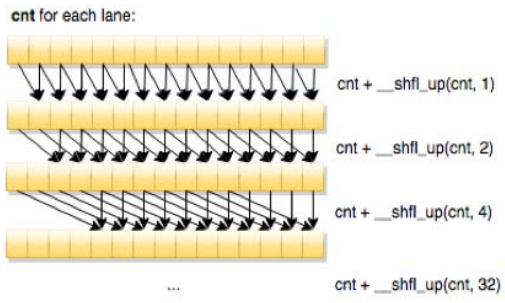

Figure 7: Phase 3b: perform prefix sum on cnt values

As a result, three indexes were produced – one for the 1024-element group (i.e. global_index), an index for each 32-element subgroup and since we have saved all predicate values – individual index within this 32- element subgroup can be calculated. This information essentially allows us to determine the indices of the valid elements in the destination array (Figure 8). Note, both kernels use grid configuration as follows: ”dim3 grid((N / 32 + block.x - 1) / block.”. The detail of the code for the Phase 3 Kernel can be found in Figure 9.

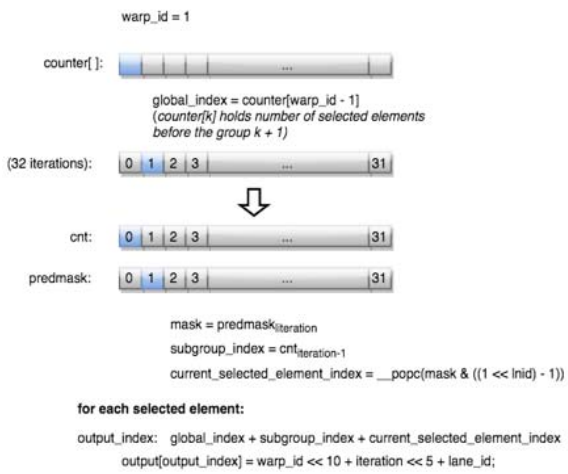

Figure 8: Phase 3c: calculate the final indexes in the output array

IV. EXPERIMENTAL RESULTS

We compared our implementation with the Thrust copy_if() method and also the Adinetz version. The following experiments were conducted on one of the nodes in the Ruby cluster provided by the Ohio Supercomputer Center.

```
/* PHASE3: produce final result array */
__global__ void alg7_kernel_phase3(int *output,
  unsigned int *counter,
  unsigned int *pred,
  const unsigned int num_items){

   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   if (tid >= (num_items >> 5) )  // divide by 32
  return;

   int lnid = lane_id();
   int warp_id = tid >> 5; // global warp number

   unsigned int predmask;
   int cnt;

   for(int i = 0; i < 32 ; i++) {
       if (lnid == i) {
           // each thread take turns to load its local var (i.e regs)
           predmask = pred[(warp_id<<5)+i];
           cnt = __popc(predmask);
       }
   }
// parallel prefix sum

#pragma unroll
   for (int offset=1; offset<32; offset<<=1) {
       int n = __shfl_up(cnt, offset) ;
       if (lnid >= offset) cnt += n;
   }

   int global_index =0 ;
   if (warp_id > 0)
       global_index = counter[warp_id -1];

   for(int i = 0; i < 32 ; i++) {
       int mask = __shfl(predmask, i); // broadcast from thr i
       int sub_group_index = 0;
       if (i > 0)
           sub_group_index = __shfl(cnt, i-1); // broadcast from thr i-1 if i>0

       if (mask & (1 << lnid ) ) // each thread extracts its pred bit
          output[global_index + sub_group_index +
            __popc(mask & ((1 << lnid) - 1))] = (warp_id<<10)+ (i<<5) + lnid  ;
   }
}
```

Figure 9: Phase 3 Kernel

The GPU used in this particular computing platform was the NVIDIA Tesla K40m, which contains 15 multiprocessors (2880 CUDA cores in total) and 12GB GDDR5 memory. A warp, the scheduling unit in CUDA, has 32 threads that perform SIMT computation on a multiprocessor. The device programs use a CUDA compiler driver 7.0.

In the first experiment, we measured the execution times for all of the algorithms by varying the number of threads per block. This allowed us to select the optimal kernel configuration for our future experiments (see Figure 10). An interesting point is that performance for the global counter version does not depend on the kernel configuration, while the algorithm using shared memory could be further optimized by choosing appropriate block size – we found that 128 threads per block work best for Tesla K40m device. This also proved to be the case for the advanced algorithm.

Note that shared memory implementation requires multiple block level synchronization barriers (__syncthreads() function) in order to avoid race conditions. This essentially makes all of the threads in the block wait for the section (before the barrier) to be completed before any threads are allowed to proceed. Thus by doubling the block size, we dramatically increase the number of threads that would potentially compete in this scenario. On the other hand, a slight decrease in performance when dipping below 128 threads per block could be explained by the fact that our

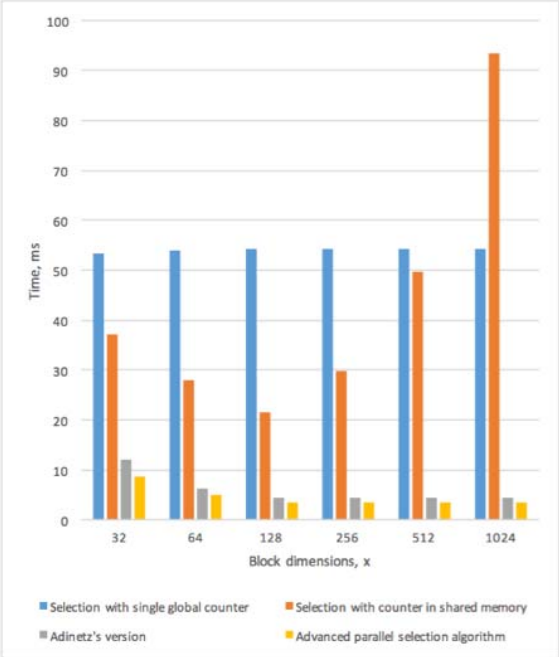

Figure 10: Performance metrics for various kernel block dimensions tested (N = 67,108,864)

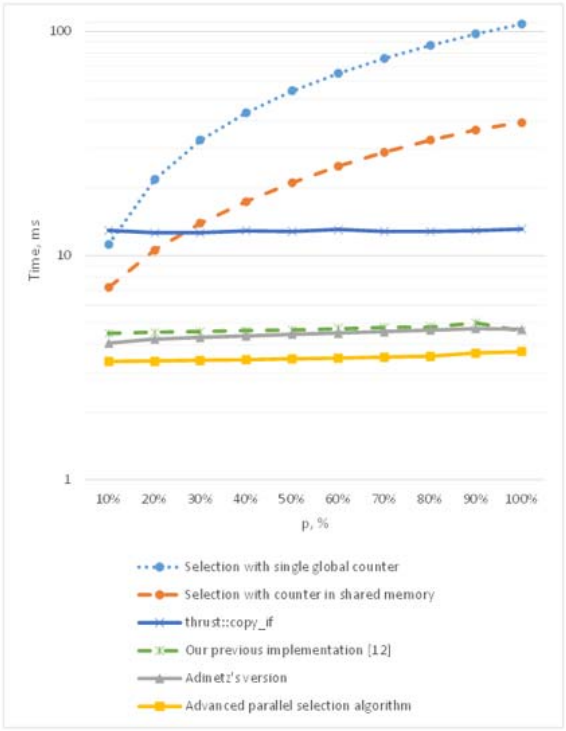

Figure 11: Performance comparison with different number of selected items

arget architecture has 4 warp schedulers, which means that there are at most 4 active warps per SMX at any given time. In other words, a block size of 128 threads will make all 4 warps schedulers busy when the block gets assigned to the processor and a block size of 32 threads would leave 2 warp schedulers out of work.

We measured the execution performance of these algorithms by varying the number of items being selected (Figure 11). We chose the block size of 128 threads. Note, for this experiment we used uniform random distribution for our data source. It can be seen that the atomic operation based approaches using single global counter and counter in the shared memory can perform better than the Thrust library only when the percentage p is very small because their execution times are proportional to the number of the selected elements. The running times of the Thrust copy_if(), Adinetz' and our both methods remain almost unchanged because they are independent of the percentage p (i.e. the number of valid elements). Furthermore, our new algorithm outperforms the Thrust copy_if(), our previous implementation in [12], and Adinetz' version. More importantly, the new algorithm can be 3.67 times faster than Thrust. Because both are list-ranking based algorithms, our method suggests a feasible improved implementation for the future version of the Thrust copy_if().

We also measured algorithm performance based on the input size. As shown in Figure 12, the performance of all

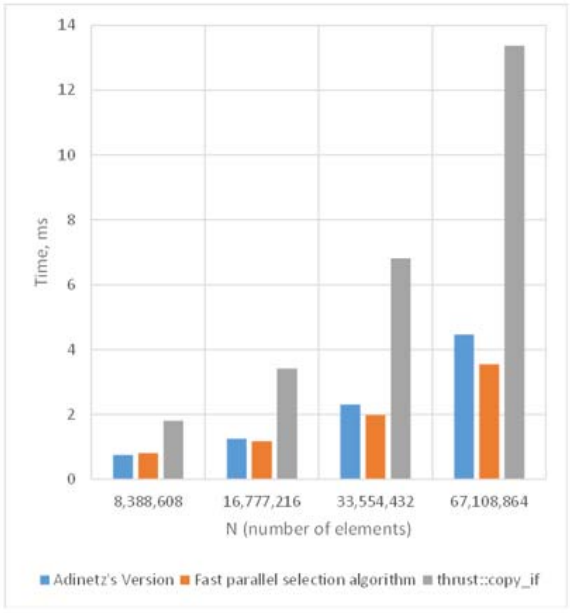

Figure 12: Performance comparison with different number of elements in the source array (p = 50%)

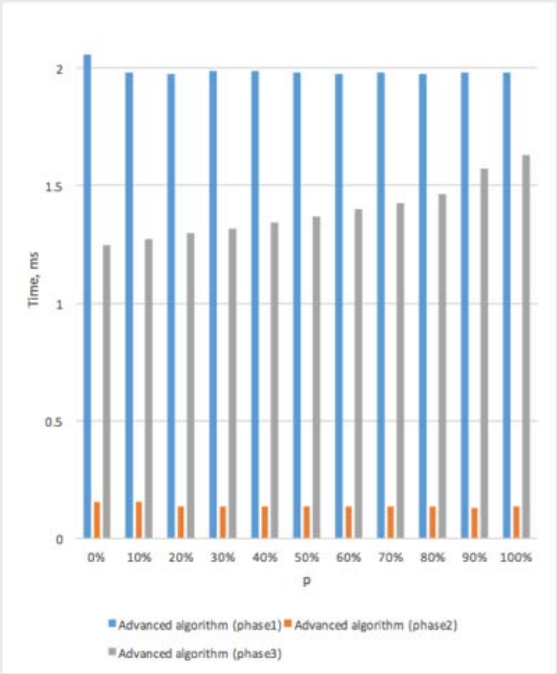

Figure 13: Execution time breakdown for the new algorithm

tested algorithms was directly proportional to the number of elements while the Thrust implementation was affected to a higher degree. As mentioned before, the Adinetz' method will issue at most one atomicAdd() for each warp and hence its execution time depends only on the number of warps (i.e. the input size / warp size), not the number of selected elements.

For our final experiment, we measured the execution breakdown times for each three phases of our improved algorithm, as illustrated in Figure 13. Phases 1 and 3 were the biggest contributors to overall performance of the algorithm, which is also why we chose not to implement the inclusive scan operation for phase 2 ourselves. As compared with the experimental result of the previous implementation reported in [12], the execution times of phase 1 and phase 3 were greatly reduced. This is because the new algorithm exploits the shuffle operations to calculate the reduction and prefix sum which result in reduced size of the intermediate counter array and fewer number of reads/writes of the intermediate array elements.

V. CONCLUSION

Our work represents an advanced implementation of a parallel selection algorithm. The experiment results are encouraging, as we were able to achieve 3.67 times better performance than what is possible using Thrust implementation. Furthermore, our new algorithm not only outperforms the Adinetz' version, but also preserves the order of the selected elements and this feature, we believe, is more important for most current applications.

ACKNOWLEDGMENTS

This research was supported by Cleveland State University 2015 Undergraduate Summer Research Award Program and by allocation of computing time from the Ohio Supercomputer Center.

REFERENCES

[1] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili, "Efficient relational algebra algorithms and data structures for gpu," Technical Report GIT-CERCS-12-01, CERCS, Georgia Institute of Technology,, Tech. Rep., 2012.

[2] S.-H. Lo, C.-R. Lee, I.-H. Chung, and Y.-C. Chung, "Optimizing Pairwise Box Intersection Checking on GPUs for Large-Scale Simulations," *ACM Trans. on Modeling and Computer Simulation*, vol. 23, no. 3, pp. 19:1–19:22, July 2013.

[3] J. Sang, C. Lee, V. Rego, and C. King, "A Fast Implementation of Parallel Discrete-Event Simulation on GPGPU," in *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, 2013.

[4] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[5] J. Hoberock and N. Bell, "Thrust: A parallel algorithms library which resembles the c++ standard template library (stl)," 2015. [Online]. Available: http://thrust.github.io

[6] Nvidia.com, "NVIDIA Kepler Compute Architecture," 2015. [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html

[7] J. C. Wyllie, "The complexity of parallel computations," PhD thesis, Cornell University, Ithaca, NY, USA, Tech. Rep., 1979.

[8] A. V. Adinetz, "CUDA Pro Tip: Optimized Filtering With Warp-Aggregated Atomics, PARALLEL FORALL," 2015. [Online]. Available: http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

[9] M. Harris and M. Garland, *Optimizing Parallel Prefix Operations for the Fermi Architecture*. San Francisco, CA, USA: Chapter 3 of the book "GPU Computing Gems - Jade Edition", Morgan Kaufmann Publishers Inc., 2011.

[10] M. Harris, "CUDA Pro Tip: Do The Kepler Shuffle, PARALLEL FORALL," 2015. [Online]. Available: http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/

[11] J. Hoberock and N. Bell, "Stream Compaction," 2015. [Online]. Available: https://thrust.github.io/doc/group__stream__compaction.html

[12] D. Bakunas-Milanowski, V. Rego, J. Sang, and C. Yu, "An Improved Implementation of Parallel Seclection on GPUs," in *Proceedings of International Symposium on Software Engineering and Applications*, 2015.